AD-A120 205 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
DETECTION OF INHERENT DEADLOCKS IN DISTRIBUTED PROGRAMS.(U)
JUN 82 K HAO, R T YEH F49620-80-C-0001
UNCLASSIFIED AFOSR-TR-82-0864 NL

END
DATE FILMED
11-82
DTIC

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ IN... BEFORE COM... ...NG FORM

1. REPORT NUMBER: AFOSR-TR- 82-0864	2. GOVT ACCESSION NO.: AD-A120205
3. RECIPIENT'S CATAL... ...UMBER	
4. TITLE (and Subtitle): DETECTION OF INHERENT DEADLOCKS IN DISTRIBUTED PROGRAMS	5. TYPE OF REPORT & PERIOD COVERED: TECHNICAL
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Kegang Hao and Raymond T. Yeh	8. CONTRACT OR GRANT NUMBER(s): F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, University of Maryland, College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Directorate of Mathematical & Information Sciences, Air Force Office of Scientific Research, Bolling AFB DC 20332	12. REPORT DATE: June 1982
	13. NUMBER OF PAGES: 6
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this paper, the concept of 'inherent deadlock' in distributed programs is defined. Several algorithms for detecting inherent deadlocks are given.

Deadlock prevention is crucial in distributed programs. In order to ensure the correctness of a distributed program, we must avoid the occurrence of the deadlock in its execution. Unfortunately, the deadlock problem in distributed program is undecidable, as the halting problem in the sequential problem. However, partial solutions to the deadlock problem exist. In this (CONTINUED)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

82 11 12 188

ITEM #20, CONTINUED: the authors shall investigate the detection of inherent deadlocks in distributed programs.

There are many models for distributed programs. In this paper, the authors shall use the model of Communicating Sequential Processes (CSP) developed by Hoare ~~[1, 2, 3]~~. In section 2 they develop some simplifications and abstractions of CSP and define the concept of 'inherent deadlock'. They they solve its decision problem.

In section 3 they authors define the concept of D-execution, and obtain a sufficient condition and a corresponding algorithm for detecting inherent deadlock.

In sections 4 and 5 the authors introduce the concept 'matching number' as the foundation for obtaining two sufficient conditions for detecting inherent deadlock. Then they reduce these conditions to the solvability of some kind of indeterminate equation and give its decision algorithm.

DETECTION OF INHERENT DEADLOCKS IN DISTRIBUTED PROGRAMS

by Kegang Hao* and Raymond T. Yeh

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

In this paper, the concept of "inherent deadlock" in distributed programs is defined. Several algorithms for detecting inherent deadlocks are given.

1. Introduction

Deadlock prevention is crucial in distributed programs. In order to ensure the correctness of a distributed program, we must avoid the occurrence of the deadlock in its execution. Unfortunately, the deadlock problem in distributed program is undecidable, as the halting problem in the sequential problem. However, partial solutions to the deadlock problem exist. In this paper, we shall investigate the detection of inherent deadlocks in distributed programs.

There are many models for distributed programs. In this paper, we shall use the model of Communicating Sequential Processes (CSP) developed by Hoare [1, 2, 3]. In section 2 we develop some simplifications and abstractions of CSP and define the concept of "inherent deadlock". Then we solve its decision problem.

In section 3 we define the concept of D-execution, and obtain a sufficient condition and a corresponding algorithm for detecting inherent deadlock.

In section 4 and 5 we introduce the concept "matching number" as the foundation for obtaining two sufficient conditions for detecting inherent deadlock. Then we reduce these conditions to the solvability of some kind of indeterminate equation and give its decision algorithm.

2. The Inherent Deadlock and Its Decision Algorithm

Some simplifications and abstractions of CSP will be given first in this section.

DEFINITION 2.1. The Commands of CSP are given as follows:

1. skip command (SK) : skip
2. assignment command (AS) : x:=e, where x is a variable and e is an expression.
3. input/output command (IO) :
   send command : A!e
   receive command : A?x

DEFINITION 2.2. The Statements of CSP are defined as follows:

1. SK statement : SK command
2. AS statement : AS command
3. IO statement : IO command
4. sequence statement : S1 ;...; Sn , where S1,...,Sn are statements
5. parallel statement : [A1:: S1 ¦¦ ... ¦¦ An:: Sn] where Si,...Sn are statements called processes, and Ai is the label of Si, i=1,...,n
6. alteration (AL) statement:

   [b1,c1 → S1 0 ... 0 bn, cn → Sn]

   where bi is a boolean expression and ci is either an SK command or an IO command. bi,ci is called a guard
7. repetition (RE) statement:

   *[b1,c1 → S1 0 ... 0 bn,cn → Sn]

DEFINITION 2.3. A Program is a statement.

EXAMPLE 1, program P:

```
[A1 :: *[b1, A2!e1 → S1] ; A2?x1  ; A3!e1'
¦¦ A2 :: *[b2, A1?x2 → S2] ; A3?x2' ; A1!e2
¦¦ A3 :: A1?x3 ; A2!e3]
```

In order to describe the execution states of a program, we introduce a special symbol @ inserted in the program. @S means that the statement S is ready for execution, and S@ means that the execution of S has terminated.

DEFINITION 2.4 A configuration is a program to which a number of special symbol @ have been inserted.

DEFINITION 2.5. The following formulas are called simple formulas:

1. @S => S@, where S is an AS or SK statement.
2. @[A1:: S1 ¦¦ ... ¦¦ An:: Sn]
   => [A1::@S1 ¦¦ ... ¦¦ An:: @Sn]
3. [A1:: S1@ ¦¦ ... ¦¦ An:: Sn@]
   => [A1:: S1 ¦¦ ... ¦¦ An:: Sn]@

*On leave from Dept. of Computer Science, China Northwestern University, Xian, Shaanxi, China.

4. S1@ ; S2 => S1 ; @S2
5. @[... 0 bi,skip -> Si 0 ...]
   => [... 0 bi,skip -> @Si 0 ...]
6. [... 0 bi,ci -> Si@ 0 ...]
   => [... 0 bi,ci -> Si 0 ...]@
7. @*[... 0 bi,skip -> Si 0 ...]
   => *[... 0 bi,skip -> @Si 0 ...]
8. @*[... 0 bi,ci -> Si 0 ...]
   => *[... 0 bi,ci -> Si 0 ...]@
9. *[... 0 bi,ci -> Si@ 0 ...]
   => @*[... 0 bi,ci -> Si 0 ...]

DEFINITION 2.6 The following formulas are called communication formulas relative to the corresponding IO commands:

1. @S => S@ , where S is an 0 statement
2. @[... 0 bi,ci -> Si 0 ...]
   => [... 0 bi,ci -> @Si 0 ...]

where ci is an IO command

3. @*[... 0 bi,ci -> Si 0 ...]
   => *[... 0 bi,ci -> @Si 0 ...]

where ci is an IO command

Definition 2.7. Two commands are called matching commands, if the following conditions are satisfied:

1. One of them is a send command, and the other is a receive command,
2. Each command of them is within the process addressed by the other command respectively,
3. The variable x and the expression e are of the same type.

For instance, in example 1 A2!e1 in A1 and A1?x2 in A2 are matching commands, A3!e1' in A1 and A1?x3 in A3 are matching commands, and so on.

DEFINITION 2.8. A configuration v is said to be deduced from u, denoted by u]== v , if either of the following conditions is satisfied:

1. v is the result of replacing all occurrences of A in u by B , where A => B is a cimple formula,
2. There are two matching IO commands in u , and v is obtained from u by simultaneous application of the communication formulas relative to these two commands.

DEFINITION 2.9 A configuration u is said to be a last configuration, if there does not exist a configuration v such that u]== v .

DEFINITION 2.10 A sequence of configurations u1,...,un is said to be an execution of a program P , if the following conditions are satisfied:

1. u1 = @P
2. ui]== ui+1 , i = 1,...,n-1
3. un is a last configuration.

Let us consider the configurations of the program in example 1:

```
u1 :
  @[ A1 :: *[ b1, A2!e1 -> S1 ] ; A2?x1 ; A3!e1'
  ¦¦ A2 :: *[ b2, A1?x2 -> S2 ] ; A3?x2' ; A1!e2
  ¦¦ A3 :: A1?x3 ; A2!e3 ]

u2 :
  [ A1 :: @*[ b1, A2!e1 -> S1 ] ; A2?x1 ; A3!e1'
  ¦¦ A2 :: @*[b2, A1?x2 -> S2 ] ; A3?x2 ; A1!e12
  ¦¦ A3 :: @A1?x3 ; A2!e3 ]

u3 :
  [ A1 :: *[ b1, A2!e1 -> @S1 ]; A2?x1 ; A3!e1'
  ¦¦ A2 :: *[b2, A1?x2 -> @S2 ]; A3?x2' ; A1!e2
  ¦¦ A3 :: @A1?x3 ; A2!e3 ]

u4 :
  [ A1 :: *[b1, A2!e1 -> S1 @]; A2?x1 ; A3!e1'
  ¦¦ A2 ::*[b2, A1?x2 -> S2 @]; A3?x2' ; A1!e2
  ¦¦ A3 :: @A1?x3 ; A2!e3 ]

u5 :
  [ A1 :: *[b1, A2!e1 -> S1] @; A2?x1 ; A3!e1'
  ¦¦ A2 :: *[b2, A ?x2 -> S2] @; A3?x2' ; A1!e2
  ¦¦ A3 :: @A1?x3 ; A2!e3 ]

u6 :
  [A1 :: *[b1, A2!e1 -> S1] ; @A2?x1 ; A3!e1'
  ¦¦ A2 :: *[b2, A1?x2 -> S2] ; @A3?x2' ; A1!e2
  ¦¦ A3 :: @A1?x3 ; A2!e3 ]
```

By the definition the following sequences are executions:

u1]== u2]== u5]== u6 .
u1]== u2]== u3]== u4]== u2]== u5]== u6 .
u1]== u2]== u3]== u4]== ...
]== u3]== u4]== u2]== u5]== u6 .

DEFINITION 2.11 A statement S in program P is called an inherent deadlock statement, if S@ does not occur in any execution of P .

DEFINITION 2.12. A program P is called an inherent deadlock program if it is an inherend deadlock statement.

Intuitively, inherent deadlock is a property of a program that deadlock always occurs in its execution. In CSP, deadlock means nontermination, i,e., either infinite loop execution, or some process is forever blocked at an IO statement.

We shall give a necessary and sufficient condition for inherent deadlock and a corresponding decision algorithm.

DEFINITION 2.13. Let P be a program. The canonical sequence of P is a sequence C1,...Cn of sets of configurations constructed as follows:

1. C1 = @P
2. If cK =/= ∅ , then
   Ck+1 = {u | there exists uk in Ck such that uk]== u and u is not in C1,...,Ck}

THEOREM 2.1. The canonical sequence is a finite sequence, that is, there exists a number n such that Cn = ∅ .

PROOF Since for every i , j Ci n Cj = ∅ , and the

number of configurations of P is finite, therefore, the canonical sequence must be a finite sequence.

The following theorem gives a necessary and sufficient condition for the occurrence of inherent deadlock. The proof is straightforward by induction and hence is omitted.

THEOREM 2.2 S is an inherent deadlock statement if and only if S@ does not occur in any configuration of the canonical sequence C1,...,Cn.

From Theorem 2.1 and Theorem 2.2 we obtain the following decision algorithm for determining whether a statement is an inherent deadlock statement.

ALGORITHM I

1) Construct the canonical sequence of P .
2) Check all configurations in it.
3) If S@ occurs in some configuration, then we know that S is not an inherent deadlock statement.
4) Otherwise, S is an inherent deadlock statement.

For example, the canonical sequence of the program in example 1 is constructed as follows:

C1={u1} C2={u2}, C3={u3,u5}, C4={u4,u6}, C5={ }.

It is clear that P@ does not occur in any configuration of the canonical sequence, so P is an inherent deadlock program.

In the decision algorithm it is necessary to check all possible configurations in its canonical sequence. However, some programs have so many such configurations that to check them exhaustively is very difficult, if it is not impossible. Therefore, developing some algorithms for detecting inherent deadlock which requires less time and space will be our goal in the rest of the paper.

3. D-execution of Program

In this section, another sufficient condition for inherent deadlock and a more efficient algorithm for detecting the inherent deadlock than algorithm 1 are developed. We shall introduce the notion of d terministic execution (D-execution) so that every p_ogram has only one such execution.

DEFINITION 3.1 A D-configuration is a program in which a number of special symbol @ have been inserted and some IO commands have been underlined.

DEFINITION 3.2 The following formulas are called D-formulas:

1,2,3,4, are the same as the simple formulas 1,2,3,4 .
5, @ r => r @ , where r is an IO command,
6. @[b1,c1 -> S1 0 ... 0 bn,cn -> Sn] => [b1,@c1 -> S1 0 ... 0 bn,@cn -> Sn]]
7, @ -> => -> @
8, [...0 bi,ci -> Si@ 0 ...]] => [... 0 bi,ci -> Si 0 ...]@
9, @*[b1,c1 -> S1 0 ... 0 bn,cn -> Sn] => *[b1,@c1 -> S1 0 ... 0 bn,@cn -> Sn]@

DEFINITION 3.3. Suppose q and r are two IO commands in a program P. q is said to precede r if either of the following conditions is satisfied:

1. There is a sequence statement S1;...;Sn (n>1) in P and q is in S1, r is in Sn
2. There is an AL (or RE) statement [... 0 bi,ci -> Si 0...] (or *[...0 bi,ci -> Si 0...]) in P and q is ci, r is in Si for some i .

DEFINITION 3.4 Suppose q and r are IO commands in a D-configuration. Command r is said to be a prime matching command of q if

1. r and q are matching commands, and
2. r is not underlined, and
3. either r is not preceded by any IO command which matches o and is not underlined, or q is i: an RE statement which does not contain r .

DEFINITION 3.5 A D-configuration v is said to be deduced from another D-configuration u , denoted by u |== v , if v is obtained from u by the application of following two steps:

1. (REPLACING) For every D-formula A => B where A occurs in u, replace all occurrences of A in u by B.
2. (UNDERLINING) If @Q occurs in u, where q is an 0 command, underline all prime matching commands of q.

Obviously, we have the following theorem:

THEOREM 3.1 If u |== v and u |== w , then v = w

DEFINITION 3.6 A D-configuration u is said to be a last D-configuration if there is no D-configuration v such that u |== v .

DEFINITION 3.7 A sequence of D-configurations u1,...,un is said to be a D-execution of a program P , if the following conditions are satisfied:

1. u1 = @P ,
2. ui |== ui+1 i=1,...,n-1 ,
3. un is a last D-configuration.

By theorem 3.1 and above definitions it is clear that the D-execution is unique for any given program.

EXAMPLE 2 Program P:

```
[A1:: [b1, A2!e1 → A2?x1 ; A3!e1
      0 b1', A2!e1' → A2?x1' ; A3!e1']
```

```
    ||A2:: [b2, A1?x2 → A3?x2' ; A1!e2
           0 b2', A1?x2' → A3x2 ; A1!e2' ]
    ||A3:: A1!x3 ; A2!e3 ]

u1 : P

u2 :
    [A1:: @ [b1, A2!e1 → A2?x1 ; A3!e1
           0 b1', A2!e1' → A2?x1' ; A3!e1']
    ||A2:: @[b2, A1?2 → A3?x2' ; A1!e2
           0 b2', A1?x2' → A3x2 ; A1!e2']
    ||A3:: @ A1?x3 ; A2!e3]

u3 :
    [A1:: [b1, @ A2!e1 → A2?x1 ; A3!e1
           0 b1', @ A2!e1 → A2?x1' ; A3!e1']
    ||A2:: [b2, @ A1?x2 → A3?x2' ; A1!e2
           0 b2', @ A1?x2' → A3x2 : A1!e2']
    ||A3:: @ A1?x3 ; A2!e3]

u4 :
    [A1:: [b1, A2!e1 → @ A2?x1 ; A3!e1
           0 b1', A2!e1' → @ A2?x1' ; A3!e1']
    ||A2:: [b2, A1?x2 → @ A3?x2' ; A1!e2
           0 b2', A1?x2' → @ A3x2 ; A1!e2']
    ||A3:: @ A1?x3 ; A2!e3]
```

From definition we know that

u1 |== u2 |== u3 |== u4

is a D-execution. Since P@ does not occur in it, the program P is an inherent deadlock program, as shown below.

LEMMA 3.1. Suppose sequence ui,...un is an execution of the program P and S is a statement in P

(1) If @S occurs in some ui, then @S also occurs in D-execution of P .

(2) If S@ occurs in some ui , then there is S'@ which occurs in the D-execution of P , where S' is the same as S or is obtained from S by inserting some @ and/or underlining some IO commands.

Lemma 3.1 can be proved in a straightforward fashion using induction. Hence, the proof is omitted here.

THEOREM 3.2.

If S is a statement in a program P , and no S'@ occurs in the D-execution, where S' either is identical to S , or is obtained from S by inserting some @ and/or underlining some IO commands, then S is an inherent deadlock statement.

PROOF. If S is not inherent deadlock statement, then S@ occurs in an execution of P . By Lemma 3.1 S'@ occurs in D-execution of P , contradicting the assumption of the theorem.

The following theorem follows directly and hence its proof is omitted.

THEOREM 3.3. If no P'@ occurs in D-execution, where P' is the same as P or is obtained from P by inserting some @ and/or underlining some IO commands, then P is an inherent deadlock program

We now give a detection algorithm derived from the above results.

ALGORITHM II.

1. Construct the D-execution of program P .

2. Check the D-execution. If no S'@ occurs in the D-execution, then S is an inherent deadlock statement and especially, if no P'@ occurs in the D-execution, then P is an inherend deadlock program.

It is clear that algorithm II is far more efficient than algorithm I. In order to illustrate this fact, we make a rough estimate as follows. Suppose that in a program there are n nondeterministic steps and each step has k (>=2) possible choices. In algorithm II only n D-configurations need to be constructed for these steps, while in algorithm I what need to be constructed are k^n configurations. So the complexity of algorithm I is an exponential function of n, whereas the complexity of algorithm II is a linear function of n.

4. Matching Number Set

Algorithm II is based on the analysis about reachibility of IO commands in the execution, but in some programs the occurrence of the deadlock is not like this. For example, the inherent deadlock of the following program cannot be discovered by algorithm II. In this program process A1 wants to send an odd number of values to A2, but A2 can only receive an even number of them.

EXAMPLE 3.

```
[A1:: A2!e1 ; *[b1, A2!e1' → A2!e1" ; S1]
||A2:: *[b2, A1?x2 → A1?x2' ; S2
       0 b2, A1?x2 → A1?x2' ; A1?x2" ; A1?2"' ;
         S2']]
```

In order to develop an algorithm for detecting this kind of deadlock, let us introduce some defintiions.

DEFINITION 4.1. If in the definition of execution we don't restrict that the application of communication formulas is only for matching IO commands, in other words, we replace definition 2.8 (2) by (2') as follows:

(2') For one or two IO commands in u, v is obtained from u by application of relative communication formulas, then we obtain a new definition about the execution, we call it V-execution. Obviously, every execution is a V-execution.

DEFINITION 4.2. Let S be a statement in a process and q be an IO command in another process, and r1,...,rn be IO commands in S which matches q . For a particular V-execution E, the number of applications of the relative communication formulas to the ri is called matching number of q in S

with respect to E , denoted by n(S,q,E) . For all possible E the set of matching numbers of q in S is called matching numbers set of q in S, denoted by N(S,q) .

Suppose that P is a parallel statement, and Q , R are processes in it, and q (inQ) and r (in R) are matching commands. It is obvious that if P is not an inherent deadlock statement then for some V-execution E, n(Q,r,E) = n(E,Q,E) . Thus, we obtain the following theorem:

THEOREM 4.1.

If N(Q,r) n N(R,q) = ∅ , then S is an inherent deadlock statement.

We shall discuss how to compute N(S,q) and how to decide whether or not N(Q,r) n N(R,q) = ∅.

DEFINITION 4.3. Let A1 , A2 be sets of natural numbers. The addition and closure operation are defined as follows:

A1 + A2 = {a | a = a1 + a2 , a1 in A1 , a2 in A2}
*A = {a | a = a1 +...+ ak , ai in A, K >= 0}
(for k=0 , we say a = a1 +...+ ak = 0)

THEOREM 4.2. Let S be a statement in a process and q be an IO command in another process. Then N(S,q) can be computed by induction on the structure of S as follows:

1. if S is a SK or AS statement, then N(S,q) = {0}
2. if S is an IO statement, then

$$N(S,q) = \begin{cases} \{1\} , \text{ if S and q are matched,} \\ \{0\} , \text{ otherwise} \end{cases}$$

3. if S = [A1:: S1 ‖ ... ‖ An:: Sn] , then $N(S,q) = \sum_{i=1}^{n} N(Si,q)$,
4. if S = S1 ;...; Sn , then $N(S,q) = \sum_{i=1}^{n} N(Si,q)$,
5. if S = [... 0 bi,ci → Si 0 ...] , then $N(S,q) = \bigcup_{i=1}^{n} (N(ci,q) + N(Si,q))$,
6. if S = *[... 0 bi ci → Si 0 ...] , then $N(S,q) = *(\bigcup_{i=1}^{n} (N(ci,q) + N(Si,q)))$

THEOREM 4.3. Every matching numbers set N(S,q) can be expressed in the following form called standard form:

$$N(S,q) = \bigcup_{i=1}^{n} (\{ai\} + *Ai) \qquad (4.1)$$

where ai is natural number, Ai is finite set of natural numbers.

PROOF. By theorem 4.2 every matching numbers set is obtained from sets {1} and {0} by finite times of the application of addition, union and closure operation. Obviously, {1} and {0} can be expressed in standard form. If we can show that the collection of all sets which can be expressed in form (4.1), denoted by I, is closed under these three operations, then the theorem will be shown.

LEMMA 4.1. I is closed under the addition operation, the union operation and the closure operation.

PROOF. This lemma follows directly from the following formulas:

*A + *B = *(A ∪ B)
*({a} + *A) = *({a} ∪ A)

THEOREM 4.4.

({a} + *A) n ({b} + *B) =/= ∅

if and only if

alx1 +...+ anxn -bly1 -...- bmym = b-a, (xi,yi >= 0)

has a natural number solution, where A and B are the finite sets of natural numbers:

A = {a1,...an} , B = {b1,...,bm} , m,n >= 0

a and b are natural numbers, b >= a .

PROOF. By the definition of closure operation, this theorem is obvious.

THEOREM 4.5. From number theory we know that

$$a_1x_1 +...+ a_nx_n - b_1y_1 -...- b_my_m = c \qquad (4.2)$$
$$(a_i,b_i > 0 , m,n >= 1 , c >= 0 , x_i,y_i >= 0)$$

has a natural number solution, if and only if the greatest common divisor (a1 ,..., an , b1,...bm) = k is a divisor of c.

We now obtain another algorithm for detecting the inherent deadlock:

ALGORITHM III. Suppose S is a parallel statement in a program P .

1. For every two matching IO commands q in Q and r in R, where Q and R are processes in S, compute N(Q,r) and N(R,q).
2. Change them into the standard form:

$$N(Q,r) = \bigcup_{i=1}^{n} (\{ai\} + *Ai)$$
$$N(R,q) = \bigcup_{i=1}^{m} (\{bi\} + *Bi) \quad .$$

3. For all i , j check

$$(\{ai\} + *Ai) \cap (\{bj\} + *Bj) = \emptyset \qquad (4.3)$$

that is, decide whether relative indeterminate equations have natural number solutions. If for

all i , j (4.3) holds, then

$$N(Q,r) \cap N(R,q) = \emptyset \qquad (4.4)$$

4. If there are Q,R,q,r such that (4.4) holds, then S is of inherent deadlock.

For instance, the matching numbers sets of the program in example 3 are:

$$N(A2, A2!e1) = *\{2, 4\}$$
$$N(A1, A2?x2) = \{1\} + *\{2\}$$

Since (2, 4, 2) = 2 is not divisor of 1,

$$2x1 + 4x2 - 2v1 = 1$$

is unsolvable. Thus the program in example 3 is an inherent deadlock program.

Algorithm III is also more efficient than algorithm I. The number of the matching numbers sets is equal to the number of matching commands in the program, which has nothing to do with the number of nondeterministic steps. So the more nondeterministic steps are there in a program, the more clear the efficiency of algorithm III is, comparing with algorithm I.

5. Algorithm IV

Let us examine an example:

EXAMPLE 4

```
[A1:: A2!e1 ; *[b1, A2?x1 → A2!e1]
||A2:: *[b2, A1?x2 → A1!e2
     0 b2',A1!e2 → A1?x2 ; A1!e2 ; A1?x2']]
N(A1, A1?x2) ∩ N(A2, A2!e1)
        = ({1} + *{1}) ∩ (*{1,2}) =/= ∅
N(a1, A1!e2) ∩ N(A2, A2?x1)
        = *{1} ∩ *{1,2} =/= ∅
```

But this program is an inherent deadlock program as shown below using Algorithm IV.

DEFINITION 5.1. Let S be a statement in a process R , Q be another process and E be a V-execution. Suppose that r1,...,rn are IO commands in S such that for every ri there is an IO command q in Q which matches ri . For a V-execution E the number of applications of the relative communication formulas to the ri is called matching number of Q in S with respect to E . For all possible E the set of matching numbers of Q in S is called matching numbers set of Q in S,denoted by N(S,Q) .

Similarly, we obtain the following theorem and algorithm IV:

THEOREM 5.1.

Suppose that P is a parallel statement and Q , R are processes in it. If

$$N(Q,R) \cap N(R,Q) = \emptyset$$

then P is an inherent deadlock statement.

The way to compute N(S,Q) is almost the same as to compute N(S,q) . The only difference is that (2) of theorem 4.2 is replaced by

(2') If S is an IO statement then

$$N(S,Q) = \begin{cases} \{1\} & \text{if there is an IO command in } Q \\ & \text{which matches } S \ , \\ \{0\} & \text{otherwise .} \end{cases}$$

Similarly, algorithm IV is also more efficient than algorithm I. Now we compute the matching numbers sets of the program in example 4 as follows:

$$N(A1,A2) = \{1\} + *\{2\} \ ,$$
$$N(A2,A1) = *\{2,4\} \ .$$

From $(\{1\} + *\{2\}) \cap (*\{2,4\}) = \emptyset$ we know that this program is an inherent deadlock program.

As we mentioned before, the algorithms (II)-(IV) can not be used to decide whether a program is an inherent deadlock program, as algorithm I does. However, their capability of detecting deadlocks are still rather strong. Several distinct kinds of inherent deadlock errors can be detected by them respectively. Especially, they are more efficient than algorithm I. So the authors consider it desirable to develop some tools for the static analysis distributed programs using these algorithms.

ACKNOWLEDGMENTS

The authors wish to thank Dr. B. Chen, Dr. J. Reed, Mr. G. Luckenbaugh, Prof. P. Yuan and Mr. B. Zhou for their comments and suggestions.

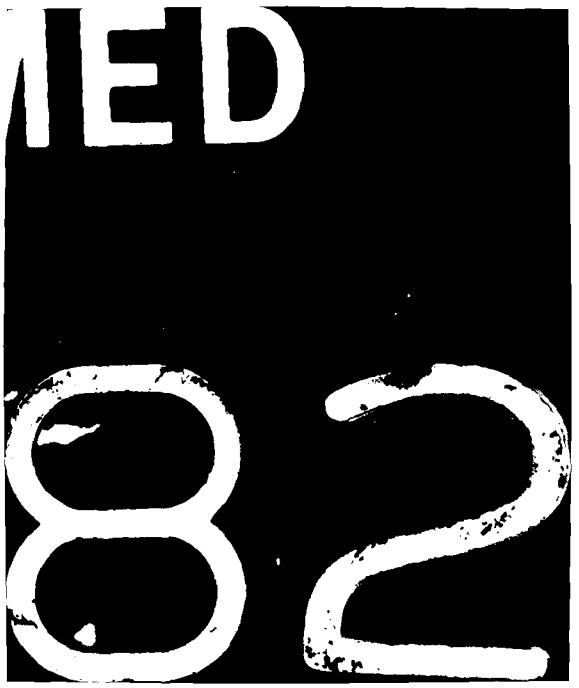
This work is partially supported by a contract from the U. S. Army under Contract DASG 60-82-C-0006 and by the Air Force under Contract F49620-80-C0001.

REFERENCES

1. Hoare, C. A. R., Communicating Sequential Processes. CACM 21, 666-677 (1978).

2. Levin, G. M. and Gries, D., A Proof Technique for Communicating Sequential Processes. Acta Informatica 15, 281-302 (1981).

3. Apt, K. R., Francez, N. and De Roever, W. P., A Proof System for Communicating Sequential Processes. ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, July 1980.


MED
82